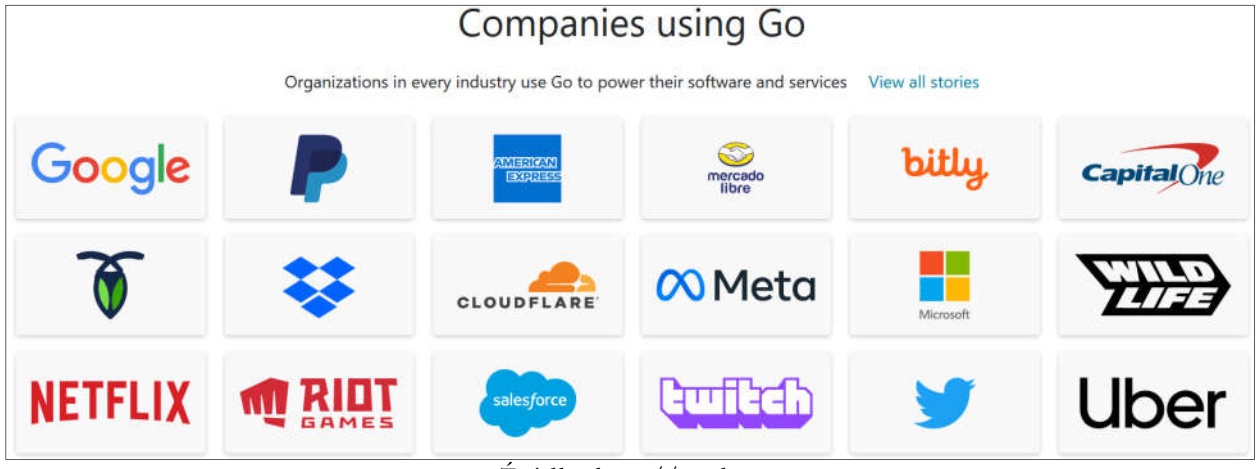
Tworzenie Aplikacji Internetowych
Laboratorium 10

Technologie po stronie serwera – Język Go - Wstęp

Język Go jest kompilowanym językiem o wysokiej wydajności, w którym m.in. w łatwy sposób można uruchomić serwer HTTP. Z tego powodu jest on często wykorzystywany przez różne firmy, co można zobaczyć na oficjalnej stronie języka - http://go.dev:



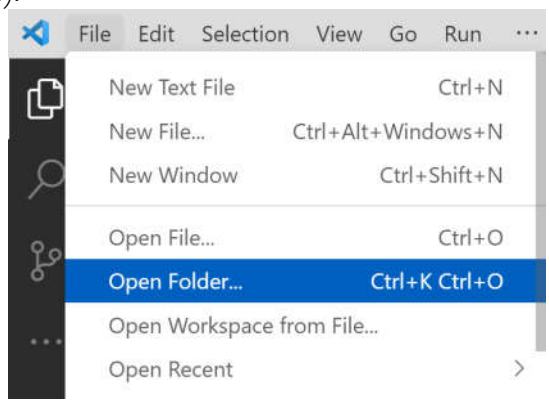
Źródło: http://go.dev

W ramach tej instrukcji zostaną przedstawione podstawy tego języka. Zbudowanie serwera przewidywane jest na kolejne instrukcje laboratoryjne.

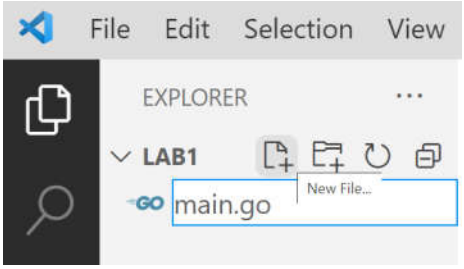
Zadanie 1

Celem zadania jest przetestowanie działania języka GO.

1. Uruchomić Visual Studio Code, utworzyć i wybrać dowolny folder roboczy projektu (File → Open Folder …):



2. W folderze utworzyć plik main.go, w razie konieczności zainstalować wszystkie proponowane dodatki przez Visual Studio Code:



3. Język Go operuje na pakietach i modułach, podstawowym pakietem jaki będzie tworzony w ramach laboratorium jest pakiet *main*, który powinien zawierać definicję funkcji startowej również o nazwie *main*. Definicja funkcji i najprostszy przykład programu wygląda następująco:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world!")
}
```

4. Warto zauważyć że środowisko Visual Studio Code i rozszerzenia do języka Go, automatycznie załączają i usuwają wykorzystywane biblioteki (jak bibliotekę fmt w powyższym przykładzie).

5. Aby uruchomić powyższy kod warto jeszcze określić nazwę modułu który tworzymy. Można zrobić to w terminalu (jeżeli terminal nie jest widoczny należy wybrać z górnego menu pozycję Terminal → New lub użyć skrótu Ctrl + `):

```
go mod init lab10
```

6. Aby przetestować działanie kodu z punktu trzeciego należy w uruchomionym terminalu skompilować i uruchomić kod za pomocą następującej komendy:

```
go run main.go
```

7. W języku go tworzone oprogramowanie (moduł) dzielone jest na paczki. Paczki mogą zawierać wydzieloną funkcjonalność, nie są one związane ani ze strukturami ani z klasami. Paczki określają foldery z plikami .go które się w nich znajdują. Pliki te widzą nawzajem wszystkie zmienne oraz funkcje z danej paczki. **Inne paczki (inne foldery) widzą tylko funkcje, struktury i zmienne których nazwy zaczynają się z dużej litery.** Ilustruje to poniższy obrazek:

var nazwa *typ* **var** nazwa *typ* = wartość **var** nazwa = wartość nazwa := wartość **var** i, j, k = wartość1, wartość2, wartość3 i, j, k := wartość1, wartość2, wartość3 **var** nazwa *complex64* = *complex*(a, b) const X *typ* = wartość	deklaracja zmiennej deklaracja zmiennej z wartością bez podania typu (wartość musi być podana) wersja skrócona deklaracja trzech zmiennych (typ może być różny) wersja skrócona (typ może być różny) liczba zespolona stała
int8, int16, int32, int64, uint8, uint16, uint32, uint64, int, uint, rune (int32), byte (uint8), float32, float64, complex64, complex128, bool, string	typy zmiennych
if warunek { … }	instrukcja if
if warunek { … } **else if** warunek { … } **else** { … }	przykład: x := true if x { fmt.Println("spełniony") } else { fmt.Println("niespełniony") }
switch zmienna { **case** wartość1: … **case** wartość2: … **default**: … }	instrukcja switch w języku Go tylko jedna instrukcja może spełniać warunek – stąd słowo break nie jest wymagane
for { … }	pętla nieskończona (wyjście za pomocą break)
for i := start; warunek; instrukcje { … }	pętla for
for j := **range** tablica { … }	pętla iterująca kolekcję (array, slice, map, string, channel)
func nazwa(par1 *typ*, …) *typzwracany* { … **return** wartość }	funkcja
func nazwa(par1 *typ*, …) (*typ1*, *typ2*, …) { … **return** wartość1, wartość2, … }	funkcja zwracająca wiele parametrów
func nazwa(par1 *typ*, …) (n1 *typ1*, n2 *typ2*, …) { …, n1 = 5, n2 = 10, … **return** }	funkcja zwracająca wiele parametrów z podaniem ich nazw, w takim wypadku pod koniec funkcji wystarczy słowo return
type nazwa **struct** { parametr1 *typ1* parametr2 *typ2* … }	struktura, do nazw parametrów pisanych z małych liter można odwołać się tylko w ramach tworzonego modułu kodu, do nazw pisanych z dużych liter można odwołać się w każdym miejscu programu (dostęp publiczny)
nazwa := nazwastr{ parametr1, parametr2, … }	inicjalizacja struktury o nazwie nazwastr
func (nazwa *nazwastr) nazwafunkcji(…) { nazwa.parametr1 = … … }	funkcja operująca na danej strukturze i posiadająca dostęp do jej pól prywatnych (podobnie jak metoda)
var *nazwa typ = &zmienna	utworzenie wskaźnika i przypisania do niego adresu zmiennej
*nazwa = wartość	przypisanie do wskazywanego elementu podanej wartości
nil	wartość pusta (null), wykorzystywana przy wskaźnikach
var nazwa *nazwastr = **new**(nazwastr)	utworzenie nowej struktury i przypisanie pod wskaźnik
nazwa := &nazwastr{wartość1, wartosc2}	j.w. ale w wersji skróconej i z podanymi wartościami
var nazwa [rozmiar]*typ*	utworzenie tablicy statycznej
var nazwa [rozmiar]*typ* = [rozmiar]*typ*{wartość1, wartość2, …}	j.w. ale z inicjalizacją
nazwa **:=** [rozmiar]*typ*{wartość1, wartość2, …}	j.w. ale w wersji skróconej
nazwatab[:3] nazwatab[3:] nazwatab[2:4]	filtrowanie tablicy: elementy do 3, od 3, od 2 do 4

Zadanie 2

Celem zadania jest dokończenie funkcji zwracającej dwa parametry – maksimum oraz minimum wyszukane w tablicy podanej jako parametr. W celu realizacji zadania należy dokończyć poniższy kod źródłowy:

```
package main
import "fmt"

func minmax(tab []int) (int, int) {
    if len(tab) < 1 {
        return 0, 0
    }
    min, max := tab[0], tab[0]
    // w tym miejscu należy napisać pętlę
    // wyszukującą minimum i maksimum
    // z tablicy tab
    return min, max
}

func main() {
    numbers := []int{10, 2, 24, 13, 20}
    a, b := minmax(numbers)
    fmt.Println("Min: ", a, "Max: ", b)
}
```

Zadanie 3

Celem zadania jest praca ze zbiorami danych.

1. Utworzyć nowy folder roboczy (np. LAB10-Z3).
2. Utworzyć strukturę ***car*** zawierającą następujące pola:
   mpg (float64) // spalanie (miles per galon)
   cylinders (int) // liczba cylindrów
   displacement (float64) // pojemność
   horsepower (float64) // moc
   weight (float64) // masa
   acceleration (float64) // przyśpieszenie
   year (int) // rocznik
   origin (int) // pochodzenie
   name (string) // nazwa
3. Przetestować strukturę wklejając następujący kod w funkcję main:

```
car0 := car{18, 8, 307, 130, 3504, 12, 70, 1, "chevrolet malibu"}
car1 := car{13, 8, 351, 158, 4363, 13, 73, 1, "ford ltd"}
car2 := car{29, 4, 98, 83, 2219, 16.5, 74, 2, "audi fox"}
car3 := car{20, 6, 232, 100, 2914, 16, 75, 1, "amc gremlin"}
car4 := car{33, 4, 91, 53, 1795, 17.4, 76, 3, "honda civic"}
car5 := car{23.2, 4, 156, 105, 2745, 16.7, 78, 1, "plymouth sapporo"}
cars := []car{car0, car1, car2, car3, car4, car5 }
fmt.Println(cars)
```

4. Sprawdzić czy zmiana parametru **czwartego** samochodu w tablicy ***cars*** wpływa także na zmianę tego parametru w obiekcie ***car4***:

```
cars[4].name = "test"
fmt.Println(cars[4])
fmt.Println(car4)
```

5. Zastanowić się czy w języku Go potrzebny jest konstruktor do inicjalizacji struktury?
6. Zmodyfikować tablicę ***cars*** na tablicę **wskaźników**. Aby to zrobić należy zmienić typ tablicy cars na ***[]*car***, a zamiast obiektów car1, car2, … podać ich adres (**operator &**). Przetestować ponownie punkt nr. 4.
7. W języku Go można tworzyć oprócz standardowych funkcji, **funkcje związane z danymi strukturami**. W tym celu należy przed nazwą funkcji podać w nawiasie okrągłym wskaźnik do danej struktury. W poniższym przykładzie zaprezentowano obie wersje funkcji – zwracają one pewną wartość podobieństwa między samochodami (im wyższa wartość tym większe podobieństwo):

```
// funkcja porównująca obiekty przekazane przez parametry
func compare(first *car, second *car) float64 {
    if first == second { return 0.0 }
    similarity := 1.0
    similarity *= 1.0 - math.Abs(first.mpg-second.mpg)/40
    similarity *= 1.0 - math.Abs(first.horsepower-second.horsepower)/300
    similarity *= 1.0 - math.Abs(first.weight-second.weight)/5000
    similarity *= 1.0 - math.Abs(first.acceleration-second.acceleration)/30
    return similarity
}

// funkcja jak metoda - porównuje obiekt bieżący z podanym jako parametr
func (this *car) compare(other *car) float64 {
    if this == other { return 0.0 }
    similarity := 1.0
    similarity *= 1.0 - math.Abs(this.mpg-other.mpg)/40
    similarity *= 1.0 - math.Abs(this.horsepower-other.horsepower)/300
    similarity *= 1.0 - math.Abs(this.weight-other.weight)/5000
    similarity *= 1.0 - math.Abs(this.acceleration-other.acceleration)/30
    return similarity
}
```

8. Przetestować powyższe funkcje za pomocą następującego kodu:

```
wynik1 := compare(&car0, &car1)
wynik2 := car0.compare(&car1)
fmt.Println(wynik1, wynik2)
```

9. Znaleźć samochód najbardziej podobny do car2. Wynik zaprezentować prowadzącemu w celu zaliczenia laboratorium.
10. Pobrać z platformy e-learningowej lub strony katedry KISI plik *cars.txt* i umieścić go obok pliku *main.go*. Następnie wczytać zawartość pliku za pomocą funkcji z następnej strony instrukcji. Po wczytaniu samochodów z pliku do dowolnej zmiennej, znaleźć samochód najbardziej podobny do trzeciego z wczytanych. Wynik zaprezentować prowadzącemu w celu zaliczenia laboratorium.
11. **Zadanie na plus**
    Napisać funkcję, która znajdzie najbardziej podobny samochód do wybranego i zwróci do niego wskaźnik. Funkcja powinna przyjmować jako parametr m.in. tablicę wskaźników do samochodów (cars []*car)

12. **Zadanie na dwa plusy**

Znaleźć dwa najbardziej podobne do siebie samochody z całego pliku cars.txt

```
func loadCars() []*car {
    cars := []*car{}
    file, err := os.Open("cars.txt")
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        line := strings.Split(scanner.Text(), "\t")
        c := car{}
        c.mpg, _ = strconv.ParseFloat(line[0], 64)
        c.cylinders, _ = strconv.Atoi(line[1])
        c.displacement, _ = strconv.ParseFloat(line[2], 64)
        c.horsepower, _ = strconv.ParseFloat(line[3], 64)
        c.weight, _ = strconv.ParseFloat(line[4], 64)
        c.acceleration, _ = strconv.ParseFloat(line[5], 64)
        c.year, _ = strconv.Atoi(line[6])
        c.origin, _ = strconv.Atoi(line[7])
        c.name = line[8]
        cars = append(cars, &c)
    }
    return cars
}
```